VIA PadLock
Montgomery Multiplier
Programming Guide

Version 1.32



This is **Version 1.32** of the VIA PadLock Montgomery Multiplier Programming Guide.

© 2004-2005 VIA Technologies, Inc. All Rights Reserved.
© 2004-2005 Centaur Technology, Inc. All Rights Reserved.

VIA Technologies and Centaur Technology reserve the right to make changes in its products without notice in order to improve design or performance characteristics.

This publication neither states nor implies any representations or warranties of any kind, including but not limited to any implied warranty of merchantability or fitness for a particular purpose. No license, express or implied, to any intellectual property rights is granted by this document.

VIA Technologies and Centaur Technology make no representations or warranties with respect to the accuracy or completeness of the contents of this publication or the information contained herein, and reserves the right to make changes at any time, without notice. VIA Technologies and Centaur Technology disclaim responsibility for any consequences resulting from the use of the information included herein.

VIA and VIA C3 are trademarks of VIA Technologies, Inc.

CentaurHauls is a trademark of Centaur Technology, Inc..

Intel is a registered trademark of Intel Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

LIFE SUPPORT POLICY

VIA processor products are not authorized for use as components in life support or other medical devices or systems (hereinafter called life support devices) unless a specific written agreement pertaining to such intended use is executed between the manufacturer and an officer of VIA.

1. Life support devices are devices which (a) are intended for surgical implant into the body or (b) support or sustain life and whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.

2. This policy covers any component of a life support device or system whose failure to perform can cause the failure of the life support device or system, or to affect its safety or effectiveness.

Table of Contents

1 INTRODUCTION....2

2 ARCHITECTURE....3

2.1HARDWARE....3
2.2SOFTWARE....5
2.3MULTI-TASKING....7
2.4MONTGOMERY MULTIPLIER: THE ALGORITHM....7

3 PROGRAMMING DETAILS....9

3.1MEMORY FORMATS....9
3.2DATA REGISTERS....9
3.3REP PREFIX....11
3.4EXCEPTION HANDLING....11

4 USING MONTGOMERY MULTIPLY....13

4.1ENABLING....13
4.2STARTUP....14
4.3MANUFACTURED STATE....14

5 INSTRUCTION TIMING....15

5.1MONTMUL INSTRUCTION TIMING....15
5.2ESTIMATED RSA PERFORMANCE....16
5.3MEASURED MODEXP PERFORMANCE....16

6 SAMPLE CODE: MONTMUL....20

6 SAMPLE CODE: THE MONTGOMERY ALGORITHM....31

CHAPTER

1

INTRODUCTION

VIA provides a suite of security technologies called PadLock in all new processors, beginning with PadLock Random Number Generator in the VIA Nehemiah processor.

The VIA Esther processor adds to PadLock a number of new capabilities. This programming guide documents the VIA PadLock Montgomery Multiplier (PMM), a fast hardware large integer modulo multiplication unit to speedup modulo exponentiation in support of public-key cryptography.

Separate Programming Guides are available for PadLock Random Number Generator (RNG), PadLock Hash Engine (PHE), and PadLock Advanced Cryptography Engine (ACE).

The CPUID identification of the VIA Esther processor is:

Vendor ID:	CentaurHauls
Type:	0
Family:	6
Model:	10

For support, developers should contact ace_support@centtech.com.

CHAPTER

2

ARCHITECTURE

2.1 HARDWARE

PadLock Montgomery Multiplier (PMM) is implemented in hardware with associated microcode. The hardware performs the underlying 32*32-bit multiplication and 64+64 bit addition operations, and the associated microcode sequences these new hardware instructions so that a complete **A * B** mod **M** operation (in the Montgomery field) is performed in the context of a single x86 instruction. This section describes PMM hardware. This information is hidden from the x86 instruction interface and is not necessary for proper use of PMM instructions.

PMM makes use of the MMX logic for data loads and stores. This is similar to the use of SSE datapaths by the PadLock Advanced Cryptography Engine (ACE). However PMM is even more tightly coupled to the MMX logic than ACE is coupled to SSE as it makes use of the MMX arithmetic units in performing the actual calculation.

Figure 1 illustrates the conceptual architecture of PMM, a 128-bit wide unit added in parallel with the MMX unit sharing instruction, load, and store buses. PMM has a discrete set of internal registers that are not visible to x86 instructions.

PMM architecture includes a new microcode instruction (call it *MMSTEP*) also not visible to the x86 instruction level. The new x86 instruction (see next section) issues a carefully ordered sequence of MMSTEP microcode instructions, along with the necessary loads and stores within the MMX unit, to perform the complete modulo multiply instruction.

Once an MMSTEP instruction is ready for execution, it is dispatched to the PMM hardware by the MMX unit. The MMSTEP instructions are carefully sequenced, by the PMM microcode, along with the necessary load and store instructions in the MMX pipeline to complete the Montgomery Multiply.

Figure 1. Adding Security Features to x86 Architecture

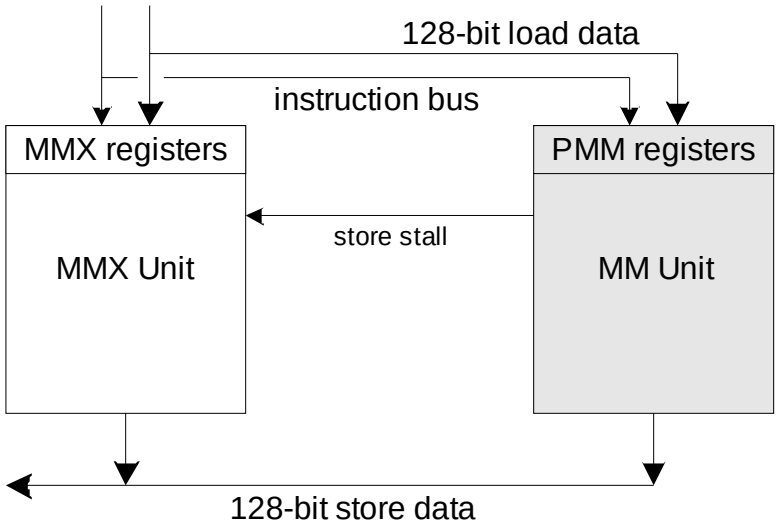


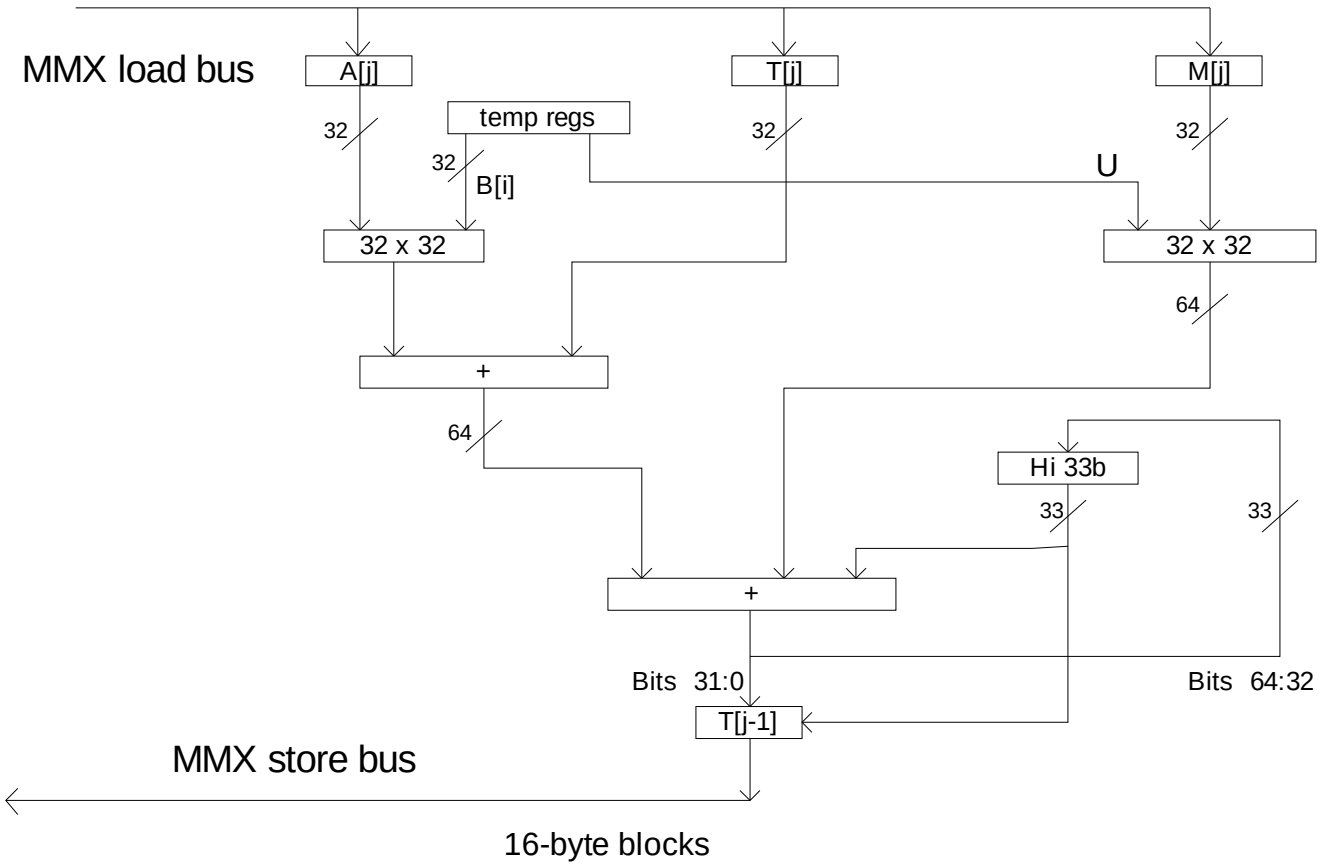
Figure 2 provides a further conceptual breakout of PMM components. There are three 128-bit input registers and one 128-bit output register. In addition, there are two internal MMX registers (TEMP REGS) used to support PMM in its calculations.

PMM is pipelined and a new calculation can be started once every two clocks. However, until we create zero-time loads and stores, the maximum effective speed of the pipelined PMM engine is to start a new calculation every 4 clocks.

The extra hardware added to the MMX unit to support PMM occupies less than 0.1 mm^2 on the processor die and runs at the processor clock frequency.

In Chapter 6 we supply sample code for using PMM. In Chapter 7 we provide illustrative C code that implements the Montgomery algorithm. The best way to understand Figure 2 is to look at this code. You are a programmer, right?

Figure 2. Conceptual PMM Hardware Overview



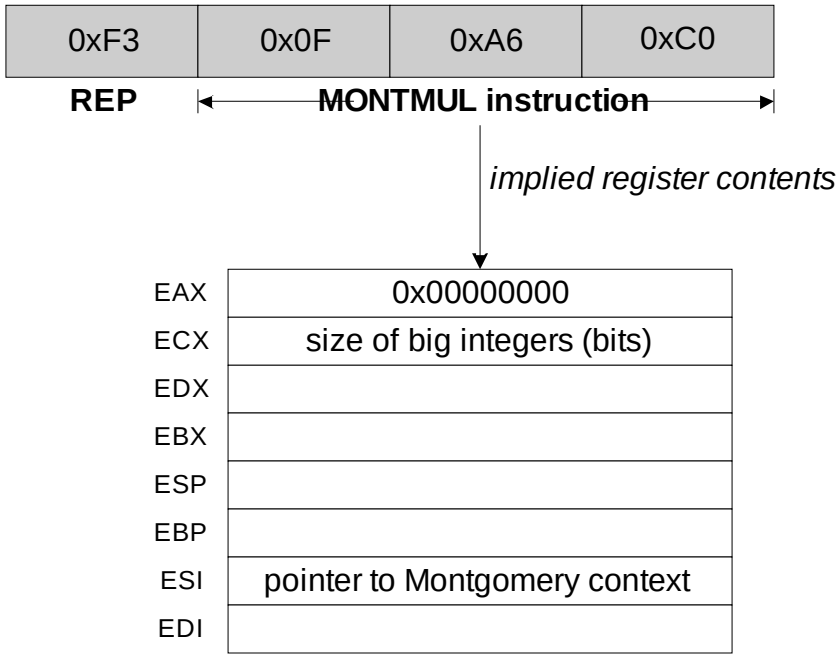
2.2 SOFTWARE

PMM is accessed using a new x86 opcode. This is a group opcode providing eight sub-opcode functions. The PMM instruction and the two PadLock Hash Engine instructions (XSHA1 and XSHA256) are accessed using three of these eight sub-opcodes. The remaining five sub-opcodes are **reserved** by VIA for new security features.

The VIA recommended name for this new instruction is **MONTMUL**. The MONTMUL instruction is not atomic; it can be interrupted just like other REP x86 instructions. However, a big integer modulo multiply is a double loop, and the outer loop REP count in ECX must be retained across interrupts until the instruction completes. For efficiency reasons, the inner loop of the multiply instruction is not interruptible, which means that the number of clocks during which the processor cannot be interrupted increases with the size of the modulus.

The MONTMUL instruction defines its operands in x86 General Purpose Registers as illustrated in Figure 3:

Figure 3. Montgomery Multiplier Instruction Format



2.3 MULTI-TASKING

There is no internal state in PMM that needs to be preserved across task switches, or whenever an interrupt or exception is taken. All state necessary to re-start the instruction at the interrupted point is preserved in either the EAX register, or in the result buffer, where the partial product is stored.

Thus PMM MONTMUL instructions neither read nor set the EFLAGS:30 bit used by PadLock Advanced Cryptography Engine and PadLock Hash Engine to support safe (and secure) multi-tasking. The instruction has no multi-tasking or security ramifications beyond those of any normal x86 instruction.

2.4 MONTGOMERY MULTIPLIER: THE ALGORITHM

The most important function in almost all public-key cryptosystems (RSA is a good example) is exponentiation modulo some modulus **M** where **M** is a very large integer, typically of length **N** = 1024 or 2048 bits (or more). That is, one must calculate

$$\mathbf{A}^{\mathbf{B}} \bmod \mathbf{M}$$

where **A** and **B** are large integers (necessarily smaller than **M**) also of length **N**.

Even ignoring the problem of reducing the result modulo **M**, this takes a long time. There are many difficult and elegant methods for reducing the number of primitive arithmetic operations required (32x32 multiplies, 64+64 adds, with a whole lot of shifting going on)

However, so long as we work with our normal, everyday integers (well, actually, a 1024-bit integer is a decimal number of more than 300 digits: the number of particles in the known universe is a number smaller than 100 digits in length. So, unless you are a cryptographer, your everyday experience does not include numbers this big!), we have to reduce each multiply operation by performing a division, as **A * A** is rather likely to be larger than **M**, and modulus is essentially the remainder after the division **(A*A) / M**.

The normal integers, mod **M,** form what is known mathematically as a *field*. Mathematicians have discovered (or invented - it's a dissertation-length discussion to explain that difference) many other fields. One of the properties of a field is that there is an operation in the field conventionally called multiplication, which is more or less comparable to ordinary multiplication.

In some cases one can transform two ordinary integers into numbers in one of these alternate fields. Then, one performs the "multiply" operation within that field, and then transforms the "product" back to ordinary integers. The result will be exactly what you would have calculated with an ordinary multiply.

This is also true of "division" in a field, since division is just multiplication by the inverse, and another property of a field is that an inverse always exists for every number.

There is a particular collection of fields known to mathematicians as Galois Fields $GF(2^K)$. Trust me; you don't need to know more, at least not as a programmer. An interesting property of these fields is that the division operation to reduce a number modulus **M** is just a binary right shift.

So it is *much* faster to perform modulo multiply of two large numbers in this field because you can replace a very slow division operation with a very fast right shift. It is even possible to construct hardware so that as you calculate the partial products of the multiply of **A * B**, the reduction modulo **M** (the right-shift) can be performed in parallel.

Many existing software implementations of $\mathbf{A}^{\mathbf{B}}$ mod **M**, in particular when used as part of a public-key cryptosystem, already make use of the Montgomery algorithm. However, there is one drawback to using the

Montgomery algorithm, which is that it is computationally expensive to calculate the "magic numbers" that are used to transform the regular integers into the Galois field. But this calculation only needs to be done once for each modulus **M**, so that when you need to calculate $\mathbf{A}^{\mathbf{B}}$ mod **M**, which involves many modulo multiplies, the gain of using the Montgomery algorithm much more than compensates for the extra time to get started.

CHAPTER

3

PROGRAMMING DETAILS

3.1 MEMORY FORMATS

All PadLock Montgomery Multiplier (PMM) data loads and stores must be 16-byte aligned. Relative to PMM notation, byte 0 is at the lowest address, byte 1 is at the next higher address, and so forth. That is, data is organized in the normal x86 little endian fashion such that PMM byte 0 is the normal x86 architecture byte 0 (the linear address provided by an instruction operand).

Thus the big integers that are pointed to in the Montgomery context must be 16-byte aligned. However, the pointer to the Montgomery context itself does not have this 16-byte alignment requirement.

3.2 DATA REGISTERS

NUMBER OF BITS IN THE BIG INTEGERS: ECX

ECX contains the number of bits in the big integers **A**, **B**, **M**, and **T**. The valid range for ECX is:

$$256 <= ECX <= 32768$$

$$ECX \bmod 128 = 0$$

At the end of instruction execution, ECX will be zero. If an interrupt (or page fault, etc.) has occurred, the value in ECX is not modified unless the instruction has completed. Information regarding the state of the interrupted instruction is stored in the EAX register.

POINTER to MONTGOMERY CONTEXT: ESI

The Montgomery context is a structure that defines the operands used in the Montgomery multiplication, and is comparable to structures found in software implementations of the Montgomery algorithm

```
typedef  struct  _mmCTX {
  uint_32        mZeroPrime;
  bigint*        A;
  bigint*        B;
  bigint*        T;                  // Result buffer: initially must be all zero
  bigint*        M;                  // Modulus
  uint_32[8]*  Scratch Buffer  //32 bytes aligned on a 16 byte boundary for internal use
} mmCTX;
```

Big integers are defined as below. Note that there is no leading "count" specifying the number of bytes, or words, or uint_32 in the big integer. For the purposes of the multiply, that information is provided in the "REP" count in ECX.

```
typedef  struct  _bigint {
uint_32 VALUE[MAX_WORDS+4];
} bigint;
```

NOTES:

It is possible for the partial product buffer to "overflow" in the Montgomery calculation. That is, the partial product can be a little bit larger than the modulus **M**. If this occurs in the final product, the software using MONTMUL will check for this and in that case the result will be **T – M.** This is well known to implementers of the Montgomery algorithm. (See the code samples in Chapters 6 and 7) What this means for PMM is that the data structure defining the big integer **T** (the partial product) must have an additional 128 bits of zeroes allocated to provide storage for the overflow should it occur.

Please see the illustrative code in Chapters 6 and 7 to learn how to calculate mZeroPrime.

MONTMUL STATE: EAX

EAX *must* be initialized to 0x00000000 at the start of a MONTMUL instruction. As a MONTMUL instruction can run for a long time (with ECX = 4096 the timing is on the order of 66,000 clocks), it supports interrupts much like other Padlock instructions like XCRYPT and XSHA. However, as large integer modulo multiplication is a double loop, the value of ECX must be retained and state for restarting the instruction contained elsewhere. The MONTMUL instruction uses EAX to retain state regarding re-starting the instruction after an interrupt has been serviced. The partial product is continuously updated in memory so as to be correct when restarting after an interrupt**.**

Except for initializing EAX = 0x00000000 at the start of the instruction, the value in EAX should be considered undefined by your program. Centaur Technology may change the way this state information is preserved in the EAX register for microcode efficiency, or to reflect improvements in underlying PMM hardware.

MONTMUL TEMP: EDX

EDX is used by the microcode of the MONTMUL instruction to store temporary data during the operation, as a precaution against certain interrupt conditions. The value in EDX is not used as input to the MONTMUL instruction, and the value is **undefined** when the instruction completes.

3.3 REP PREFIX

The REP MONTMUL instruction has only a few similarities with the x86 REP MOVS instruction:

- An address-size prefix affects the size of ESI.
- The usual address exceptions (past the segment limit, page fault, etc.) can occur.

And there are some different semantics:

- The value in ESI is unchanged as it isn't a "source" register pointer
- EDI is not used; there is no "destination" pointer
- A REPNE prefix causes undefined behavior.
- An operand size prefix causes an Invalid Instruction exception.
- The DF (direction flag) in EFLAGS has no effect.
- The full 32 bit EAX and ECX registers are used regardless of any address or operand prefix.
- The value in ECX does not change until the operation completes, when it is set to zero.

3.4 EXCEPTION HANDLING

During execution of a REP MONTMUL instruction, any number of exceptions or interrupts may occur.

For most exceptions, microcode instructions appear, logically, just like atomic x86 instructions: that is, the exception or interrupt occurs only before the instruction begins, or after it ends.

For normal x86 REP instructions, these exceptions occur "naturally" when the base instruction being repeated has completed, and after all architecture registers such as source and data pointers, and the REP counter have been updated. The MONTMUL instruction behaves similarly with respect to these controlled exceptions and interrupts.

However some exceptions occur "in the middle of" the microcode sequence. A good example is a page fault that occurs when loading a text block from ES:[ESI]. From the perspective of the MONTMUL instruction, the page fault occurs "instantaneously". Execution branches to some fault handler, and, almost always, the original REP MONTMUL instruction will be re-issued by the translator.

Thus the microcode must not change an architecturally visible register until it is guaranteed that no page fault or similar exception can occur, which would cause an incorrect state on instruction restart. This is handled by testing the memory buffers for all the big integers involved in the instruction (**A, B, T, M**) before

actually commencing any calculations. This ensures that any page fault will occur before any architecture register is changed and before any data are written to memory.

CHAPTER

4

USING MONTGOMERY MULTIPLY

4.1 ENABLING

To use VIA PadLock Montgomery Multiplier (PMM), three conditions must be met:

- PMM must physically exist on the chip. Its existence can be discovered from the Centaur Extended CPUID Feature Flags. This condition is permanently set as part of the manufacturing process.
- PMM must be enabled. The enabled state can be also discovered from the Centaur Extended CPUID Feature Flags. PMM can be enabled/disabled by writing to the Function Control Register (FCR), MSR 0x1107. The RESET default is enabled. But, see the next condition.
- SSE instructions must be enabled via the standard x86 method of enabling the FXSAVE/FXRSTOR instructions using CR4[9] This CR4 enabling also enables the full set of SSE instructions on VIA Esther and Nehemiah processors.

VIA Esther and Nehemiah processors support the Centaur Extended Feature Flags. When the CPUID instruction is executed with EAX=0xC0000000, the processor returns 0xC0000001 in EAX. Note that earlier VIA C3 processors do not support the Centaur Extended CPUID Functions and will return EAX=0xC0000000 or EAX=0x00000000 (see relevant VIA processor datasheets for details on the extended CPUID function).

If a CPUID with EAX=0xC0000000 returns a value in EAX >= 0xC0000001 then Centaur Extended Feature Flags are supported. A CPUID with EAX=0xC0000001 then returns the Centaur Extended Feature Flags in EDX. There are two bits in EDX that describe PMM:

- EDX[8] == 0 PMM does not exist on this chip. FCR[28] cannot be set to enable PMM and MONTMUL instructions cause an Invalid Opcode Fault.
- EDX[8] == 1 PMM exists. The behavior of MONTMUL instructions is dependent upon whether or not PMM is enabled.

- EDX[9] == 0 PMM is disabled. MONTMUL instructions cause an Invalid Opcode Fault. If PMM is present, FCR[28] can be set to enable PMM
- EDX[9] == 1 PMM is enabled (usually the RESET default). MONTMUL instructions behave as defined in this programming guide (provided CR4[9] =1)
- VIA Esther and Nehemiah processors implement a Function Control Register (FCR), MSR 0x1107, which can be written by software to enable or disable various features. One bit controls PMM - FCR [28]:
- FCR[28] == 0 PMM is not enabled. If EDX[8] indicates that PMM is present, a WRMSR can set this bit to 1 thus enabling PMM.
- FCR[28] == 1 PMM is enabled. A WRMSR can set this bit to 0 thus disabling PMM.

4.2 STARTUP

The RESET signal is an immediate asynchronous process: RESET halts operation immediately regardless of the state of the processor. As part of the RESET process, the following PMM actions are performed:

- Any in-progress MONTMUL instruction is immediately cancelled and undefined results may be stored.
- The FCR is reset to its default value. This may or may not directly change PMM enable status.
- The CR4 register is reset to zero. Since this resets the FXSAVE/FXRSTOR enable, (CR4[9]) PMM is indirectly disabled.

The INIT signal is an interrupt that occurs between x86 instructions: INIT never halts instructions in the middle of their execution (except for REP string instructions, where INIT behaves like INTR). The effect of INIT on PMM is:

- The CR4 register is reset to zero. Since this resets the FXSAVE/FXRSTOR enable (CR4[9]), PMM is indirectly disabled.

4.3 MANUFACTURED STATE

PMM is tested as part of the VIA processor manufacturing process. Relative to potential fault coverage, however, this level of testing is not 100% complete. For certain sensitive applications, additional testing (by the application) may be desirable.

CHAPTER

5

INSTRUCTION TIMING

5.1 MONTMUL INSTRUCTION TIMING

As for all x86 instructions that reference memory, the timing of MONTMUL instructions is non-deterministic due to cache misses, page faults, interrupts, load and store buffer stalls, SSE queue-full stalls, external DMA snoops, etc. Consistent with other published x86 instruction timings, this section ignores these variables and assumes a perfect environment: everything is in the cache, no interrupts, no snoops, no stalls, etc.

The MONTMUL instruction is particularly complex. Our measurements (in simulation) are in line with our original estimates:

Let **|M|** be the number of bits in the big integers **A**, **B**, and **M**. (Including any necessary zeros padded at the left so that **|M|** is a multiple of 128.)

Let **N** = **|M|** / 1024, that is let **N** be the number of kilobits in the big integers. Then the number of clocks **C** to execute a MONTMUL ***instruction*** is

$$\mathbf{C = 1024 * (4*N^2 + K_1*N) + K_2}$$

where

$$\mathbf{K_1 \approx 0.56 – 1.06}$$

and

$$\mathbf{K_2 \approx 35}$$

Why, you ask, is the value for $\mathbf{K_1}$ so indeterminate? There are three factors to consider:

First, the Montgomery algorithm can "overflow", as discussed in Chapter 3 and as you can see illustrated in the sample code from Chapter 7. This is when it will be necessary to subtract the modulus (one time) from the result. In this case, the calculations require one extra (partial) execution of the inner loop for every pass through the outer loop once the overflow condition occurs.

Second, although the numbers are 16-byte aligned, at the conclusion of each outer loop the 33 bits of the carry are written to memory using a 64-bit MMX store instruction, which is not 16-byte aligned, but 4-byte aligned. There is no performance penalty for this store except when the store crosses a cache-line boundary.

A good rule of thumb is that if this alignment stall occurs, the effective value of $\mathbf{K_1}$ for that set of numbers will increase by about 0.25 – 0.35 depending on the size of $|\mathbf{M}|$, with the larger increase in $\mathbf{K_1}$ occurring for smaller values of $|\mathbf{M}|$.

Fortunately, these two conditions can never both occur. Whenever the extra calculation is required, the final store of the carry bits occurs on a 16-byte aligned boundary.

The third factor affecting $\mathbf{K_1}$ is the size of the big integers $|\mathbf{M}|$. The microcode executes the inner loop of the algorithm, in the integer pipeline, in 15 clocks, while the PMM hardware and the associated MMX logic require 16 clocks for the inner loop. For smaller values of $|\mathbf{M}|$, these extra clocks for the MONTMUL calculation simply overlap some of the integer instructions necessary for the outer loop. For larger values of $|\mathbf{M}|$ the MMX pipeline will eventually fill and the integer pipeline stalls for one clock during the execution of the inner loop. In this latter case, however, the effective overlap of the integer pipeline with the MONTMUL calculation during the tail of the outer loop is at a maximum. The smallest measured value for $\mathbf{K_1}$ (0.561) occurred during a simulation using numbers with $|\mathbf{M}| = 2304$.

Of course, it is important to remember that the time to perform the MONTMUL instruction is not the total time to perform a modular multiplication.

5.2 ESTIMATED RSA PERFORMANCE

Based on the timing numbers above, we can estimate the performance of a VIA Esther processor for the RSA public-key encryption algorithm.

Assume a 2 Ghz VIA Esther processor and 1024-bit keys. Assume also that the time to perform the (many) multiplication operations dwarfs all other instruction overhead. Assume that the Chinese Remainder Theorem has been used so that the actual **modexp** calculations will be with 512-bit numbers rather than with 1024-bit numbers.

With those assumptions, a 2 Ghz VIA Esther processor will perform about 900 RSA signings per second, for 1024 bit keys. Similar assumptions for larger key lengths lead to estimates of about 130 RSA signings per second for 2048 bit keys, and about 18 signings per second for 4096 bit keys.

5.3 MEASURED MODEXP PERFORMANCE

Now, let's get back to the "real world".

While the performance of the MONTMUL instruction as measured conforms closely to the theoretical estimates given above, the overhead of the other instructions in performing a modexp calculation is not negligible, especially for small values of the modulus.

Below we show two representative outputs from the sample code of Chapter 6, which performs modexp calculations for various key lengths using both the GMP library and with code using the VIA PadLock MONTMUL instruction. These values were calculated using a VIA Esther processor running at 1 Ghz.

To understand these numbers, relative to the estimates of RSA performance in section 5.2 above, it is important to understand that the modexp time is a very close approximation of the complete RSA signing time, and that time to perform modexp a 1Ghz for a key length of N is a close approximation of the time to perform the RSA operation on a 2Ghz processor for a key length of 2N. This is because while using the Chinese Remainder Theorem to speed the RSA calculation for keys of length N means using modexp of length 1/2N, you have to perform two modexp calculations at the smaller modulus size.

As you can see, the calculated times for a modexp using the MONTMUL hardware are quite close to our theoretical estimates for RSA keys of 4096 bits (modexp 2048 bits): 17 versus 18, while the deviation is somewhat larger for RSA keys of 1024 bits (modexp 512 bits): 620 – 740 versus 900. This make sense: for larger and larger key lengths the advantage of using MONTMUL is greater as relatively more time is spent "inside" MONTMUL and less in the accompanying x86 code.

This is consistent with the performance of MONTMUL based modexp versus the GMP library. The hardware-based calculation becomes relatively more efficient as the key lengths get longer.

It is also worth remembering that the modexp calculations in our sample code, using our MONTMUL instruction, are clearly not optimized. It is the "gut feeling" of your humble author that properly optimized code using MONTMUL will be able to perform over 800 modexp per second for 512-bit moduli on a 1Ghz Esther processor. Perhaps in a later version of this document we will provide a third code sample, with carefully optimized and hand-tuned assembly language code for modexp with MONTMUL

```
LENGTH = 256 bits.
  GMP:   1550.0 ModExp/per sec
  PMM:   3100.0 ModExp/per sec

LENGTH = 384 bits.
  GMP:    442.9 ModExp/per sec
  PMM:   1550.0 ModExp/per sec

LENGTH = 512 bits.
  GMP:    206.7 ModExp/per sec
  PMM:    620.0 ModExp/per sec

LENGTH = 640 bits.
  GMP:    110.7 ModExp/per sec
  PMM:    442.9 ModExp/per sec

LENGTH = 768 bits.
  GMP:     67.4 ModExp/per sec
  PMM:    258.3 ModExp/per sec

LENGTH = 896 bits.
  GMP:     44.9 ModExp/per sec
  PMM:    172.2 ModExp/per sec

LENGTH = 1024 bits.
  GMP:     31.3 ModExp/per sec
  PMM:    114.8 ModExp/per sec

LENGTH = 1152 bits.
  GMP:     22.3 ModExp/per sec
  PMM:     88.6 ModExp/per sec

LENGTH = 1280 bits.
  GMP:     16.7 ModExp/per sec
  PMM:     63.3 ModExp/per sec

LENGTH = 1408 bits.
  GMP:     12.8 ModExp/per sec
  PMM:     49.2 ModExp/per sec

LENGTH = 1536 bits.
  GMP:     10.0 ModExp/per sec
  PMM:     39.2 ModExp/per sec

LENGTH = 1664 bits.
  GMP:      7.9 ModExp/per sec
  PMM:     30.7 ModExp/per sec

LENGTH = 1792 bits.
  GMP:      6.4 ModExp/per sec
  PMM:     24.8 ModExp/per sec

LENGTH = 1920 bits.
  GMP:      5.2 ModExp/per sec
  PMM:     20.5 ModExp/per sec

LENGTH = 2048 bits.
  GMP:      4.5 ModExp/per sec
  PMM:     17.1 ModExp/per sec
```

```
LENGTH = 256 bits.
  GMP:   1233.3 ModExp/per sec
  PMM:   3700.0 ModExp/per sec

LENGTH = 384 bits.
  GMP:    411.1 ModExp/per sec
  PMM:   1233.3 ModExp/per sec

LENGTH = 512 bits.
  GMP:    205.6 ModExp/per sec
  PMM:    740.0 ModExp/per sec

LENGTH = 640 bits.
  GMP:    108.8 ModExp/per sec
  PMM:    411.1 ModExp/per sec

LENGTH = 768 bits.
  GMP:     67.3 ModExp/per sec
  PMM:    264.3 ModExp/per sec

LENGTH = 896 bits.
  GMP:     44.0 ModExp/per sec
  PMM:    168.2 ModExp/per sec

LENGTH = 1024 bits.
  GMP:     31.4 ModExp/per sec
  PMM:    112.1 ModExp/per sec

LENGTH = 1152 bits.
  GMP:     22.4 ModExp/per sec
  PMM:     86.0 ModExp/per sec

LENGTH = 1280 bits.
  GMP:     16.7 ModExp/per sec
  PMM:     62.7 ModExp/per sec

LENGTH = 1408 bits.
  GMP:     12.8 ModExp/per sec
  PMM:     49.3 ModExp/per sec

LENGTH = 1536 bits.
  GMP:     10.0 ModExp/per sec
  PMM:     37.8 ModExp/per sec

LENGTH = 1664 bits.
  GMP:      7.9 ModExp/per sec
  PMM:     31.1 ModExp/per sec

LENGTH = 1792 bits.
  GMP:      6.4 ModExp/per sec
  PMM:     24.8 ModExp/per sec

LENGTH = 1920 bits.
  GMP:      5.2 ModExp/per sec
  PMM:     20.6 ModExp/per sec

LENGTH = 2048 bits.
  GMP:      4.5 ModExp/per sec
  PMM:     17.1 ModExp/per sec
```

CHAPTER

6

SAMPLE CODE: USING MONTMUL

```
/*
  Montgomery Multiply Test Program (MMTEST.C)
  Copyright (c) 2004-2005 Centaur Technology
  All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

 * Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.
 * Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.
 * Neither the name of Centaur Technology nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
```

```
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
*/

/*
  NOTE REGARDING THE GNU MULTI-PRECISION ARITHMETIC LIBRARY
This test program links to routines provided by the GNU Multiple Precision
Arithmetic Library (GMP). The GMP is released under the terms of the GNU
Lesser General Public License (LGPL), a copy of which may be obtained at

  http://www.gnu.org/copyleft/lesser.html

Use of the GMP is not required to use the MONTMUL instruction of the VIA
Esther processor.  However, should you choose to use code from the GMP in
support of operations using MONTMUL, be sure to observe the terms of that
license.
*/

/*
  To build this program:
    gcc -O3 -o mmtest mmtest.c -lgmp
*/
#include <stdio.h>
#include <gmp.h>
#include <time.h>

#define REPEAT    2
#define COUNTS  37

#define MAX_BITS  65536
#define MAX_WORDS ((MAX_BITS/32))

#define TRUE      1
#define FALSE     0

#define MONTMUL(TEMP, DUMMY, CONTEXT) \
```

```
  asm __volatile__ (".byte 0xF3, 0x0F, 0xA6, 0xC0\n" \
    : "=c" (TEMP), "=a" (DUMMY) \
    : "c" (TEMP), "a" (0), "S" (CONTEXT));

#define XSTORE(TEMP, DUMMY, DATA) \
  asm __volatile__ (".byte 0xF3, 0x0F, 0xA7, 0xC0\n" \
    : "=c" (TEMP), "=a" (DUMMY) \
    : "c" (TEMP), "d" (3), "D" (DATA));

typedef unsigned int   uint_32;
typedef unsigned long long uint_64;

typedef struct _mmCTX {
  unsigned int mzeroprime;
  int *A;
  int *B;
  int *D;
  int *M;
  unsigned int *scratch;
} mmCTX;

/* Size of big integers in 32-bit (int) chunks.
   Valid between 8 and 1024 in 128 bit increments. */

  int LENGTH, BLENGTH;

  uint_32 mZeroPrime;

  mmCTX CONTEXT = {0};

  uint_32 aPrime[MAX_WORDS] = {0};
  uint_32 varA[MAX_WORDS] = {0};
  uint_32 varB[MAX_WORDS] = {0};
  uint_32 varC[MAX_WORDS] = {0};
  uint_32 varM[MAX_WORDS] = {0};
  uint_32 varRM[MAX_WORDS] = {0};

  uint_32 Result[MAX_WORDS] = {0};
```

```
  uint_32 One[MAX_WORDS] = {0};
  uint_32 SCRATCH[8] = {0};


/* Inverse -modulus^(-1) mod B for odd modulus */

uint_32 bigIntInvMon(uint_32 *modulus){

  uint_32 i;
  uint_64 x = 2, y = 1;
  uint_64 mZero = (uint_64)(modulus[0]);

  for (i = 2; i <= 32; i++, x <<= 1) {
    if (x < ((mZero * y) & ((x << 1) - 1))) {
      y += x;
    }
  }
  return (uint_32)(x - y);
}

int bigIntGTE(uint_32 *A, uint_32 *B) {
  int i;

  for (i = LENGTH + 1; i >= 0; i--) {
    if (A[i] > B[i]) return TRUE;
    if (A[i] < B[i]) return FALSE;
  }
  return TRUE;
}

void bigIntSub(uint_32 *A, uint_32 *B) {
  int i,j;

  for (i = 0; i < LENGTH + 2; i++) {

    if (B[i] > A[i]) {
      j = i+1;
      A[j]--;
```

```
        while (A[j] == 0xFFFFFFFF)
          A[j++]--;
      }
      A[i] = A[i] - B[i];
    }
}

int MulMon(uint_32 *a, uint_32 *b, uint_32 *modulus, uint_32 *t) {

    int  TEMP = LENGTH << 5;
         int DUMMY = 0;
    int  i;

    uint_32 *temp = t;

    CONTEXT.A = a;
    CONTEXT.B = b;
    CONTEXT.D = t;

    for (i=0; i<8; i++)
      SCRATCH[i] = 0;

    for (i=0; i<LENGTH+2; i++)
      *temp++ = 0;

    MONTMUL(TEMP, DUMMY, &CONTEXT);

    if (bigIntGTE(t, modulus))
      bigIntSub(t, modulus);

}

    uint_32 * X;

/*
    CORNER CASE!
    If the first 32 bits of the exponent are zero this
    won't work correctly - but I'm not gonna fix it as
```

```
  as this is just a sample to illustrate USE of the
  new MONTMUL instruction, not a ModExp tutorial
*/

void ModExp() {

  int i,j;
  int index = LENGTH - 1;

  CONTEXT.mzeroprime = mZeroPrime;
  CONTEXT.M = varM;
  CONTEXT.scratch = &SCRATCH[0];

  uint_32 counter = 32;

  One[0] = 1;

  MulMon(varA, varRM, varM, aPrime);

  j = varB[index];

  while ( (j & 0x80000000) == 0) {
    counter--;
    j = ((unsigned int) j) << 1;
  }

  for (i=0; i<LENGTH+2; i++)
    Result[i] = aPrime[i];

  counter--;
  j = ((unsigned int) j) << 1;

  while (index >= 0) {
    while ( counter-- ) {

      MulMon(Result, Result, varM, varC);

      if (j & 0x80000000) {
```

```
          MulMon(varC, aPrime, varM, Result);
        }
        else {
          for (i=0; i<LENGTH+2; i++)
            Result[i] = varC[i];
        }
        j = ((unsigned int) j) << 1;
      }

      j = varB[--index];
      counter = 32;
    }
    MulMon(Result, One, varM, varC);
}


int main (int argc, char *argv[]) {

    int i,j,counter;
    int DUMMY = 0;
    int TEMP = 0;

    time_t gmp_ticks, pmm_ticks, ticks;
    float rate;

    mpz_t x,y,z,mod,product,r,temp;

    for (LENGTH = 8; LENGTH < 65; LENGTH += 4 ) {

      gmp_ticks = 0;
      pmm_ticks = 0;
          printf("LENGTH = %d bits.\n", LENGTH*32);

      for (counter = 0; counter < REPEAT; counter++) {

/* Generate a test case - get three big random numbers from
   our hardware RNG */
```

```
mpz_init2(x, 65536);
mpz_init2(y, 65536);
mpz_init2(z, 65536);
mpz_init2(r, 65536);
mpz_init2(mod, 65536);
mpz_init2(product, 65536);
mpz_init2(temp, 65536);


for(i=0; i<MAX_WORDS; i++) {
   aPrime[i] = 0;
   varA[i] = 0;
   varB[i] = 0;
   varC[i] = 0;
   varM[i] = 0;
   varRM[i] = 0;
}

X = (unsigned int *) x;
X[1] = LENGTH;

TEMP = LENGTH*4;
XSTORE(TEMP, DUMMY, X[2]);

X = (unsigned int *) y;
X[1] = LENGTH;
TEMP = LENGTH*4;
XSTORE(TEMP, DUMMY, X[2]);

X = (unsigned int *) mod;
X[1] = LENGTH;
TEMP = LENGTH*4;
XSTORE(TEMP, DUMMY, X[2]);

if (mpz_cmpabs(x, mod) > 0)
   mpz_swap(x, mod);

if (mpz_cmpabs(y, mod) > 0)
```

```
          mpz_swap(y, mod);

/* If modulus is even we gotta add 1, the Montgomery algorithm requires
   an odd modulus */
        if (mpz_even_p(mod)) {
          mpz_add_ui(z, mod, 1);
          mpz_set(mod, z);
        }

        X = (unsigned int *) x;
        X = (unsigned int *) X[2];
        varA[LENGTH] = 0;
        for (i=0; i<LENGTH; i++)
          varA[i] = *X++;

        X = (unsigned int *) y;
        X = (unsigned int *) X[2];
        varB[LENGTH] = 0;
        for (i=0; i<LENGTH; i++)
          varB[i] = *X++;

        X = (unsigned int *) mod;
        X = (unsigned int *) X[2];
        varM[LENGTH] = 0;
        for (i=0; i<LENGTH; i++)
          varM[i] = *X++;

/* Mod exp in software */
        ticks = clock();

        for (i=0; i<COUNTS; i++)
          mpz_powm(product,x,y,mod);

        gmp_ticks += clock() - ticks;

/* Use the new hardware instruction -- get RM and R (mZeroPrime)
   because the hardware needs them and we can't do it any faster */
```

```
  X = (unsigned int *) temp;
  X[1] = LENGTH + 1;
  X = (unsigned int *) X[2];
  X[LENGTH] = 1;
  for (i = 0; i < LENGTH; i++)
    X[i] = 0;

  mpz_mul(z,temp,temp);
  mpz_mod(r,z,mod);
  X = (unsigned int *) r;
  X = (unsigned int *) X[2];
  varRM[LENGTH] = 0;
  for (i=0; i<LENGTH; i++)
    varRM[i] = *X++;

  mZeroPrime = bigIntInvMon(varM);

  ticks = clock();
  for (i=0; i<COUNTS; i++)
    ModExp();

  pmm_ticks += clock() - ticks;

  X = (uint_32 *) product;
  X = (uint_32 *) X[2];

  for (i=0; i<LENGTH; i++) {

    if ( varC[i] != *X ) {
      printf ("ERROR: Results differ\n\n");
      exit (0);
    }
    X++;
  }
}

rate = ((float) (COUNTS * REPEAT * CLOCKS_PER_SEC)) / gmp_ticks ;
printf("  GMP:  %7.1f ModExp/per sec \n", rate);
```

```
    rate = ((float) (COUNTS * REPEAT * CLOCKS_PER_SEC)) / pmm_ticks ;
    printf("  PMM:  %7.1f ModExp/per sec \n\n", rate);

  }
  return 0;
}
```

CHAPTER

7

SAMPLE CODE: THE MONTGOMERY MULTIPLICATION ALGORITHM

```
/*
  Montgomery Multiply in C
  Copyright (c) 2004-2005 Centaur Technology
   All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

 * Redistributions of source code must retain the above copyright notice,
   this list of conditions and the following disclaimer.
 * Redistributions in binary form must reproduce the above copyright notice,
   this list of conditions and the following disclaimer in the documentation
   and/or other materials provided with the distribution.
 * Neither the name of Centaur Technology nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*/

/*
This code illustrates the Montgomery Multiplication algorithm. During our development we generated test cases using the **Mathematica** program, which produced output in a format like the following:

```
-------------
n=2
m=10a4201d
a=a46aa5a
b=b4a3016
k=20
rm=a65c865
c=d71a93
--------------
n=4
m=f180d8c8b1ad8f1f
a=a78c9241808a4e34
b=9c8a755b17821505
k=40
rm=5cdcb8ef8a4e7317
c=3b5ab8b2823c75b5
```

Some sample test files had more than 200,000 test cases. This is the format of the test cases in the input data file referenced by the code

Mathematica is a trademark of Wolfram Research, Inc.

```
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

#define TRUE      1
#define FALSE     0
#define MAX_BITS  4352
#define MAX_WORDS ((MAX_BITS/16))

typedef unsigned short uint_16;
typedef unsigned int   uint_32;
typedef unsigned long long uint_64;

typedef struct _bigInt {
  uint_16 length;
  uint_16 value[MAX_WORDS];
} bigInt;

int LENGTH;

/*---------------------------------------------------------------------------*/
bigInt *newBigInt(uint_32 initialValue) {
  bigInt *aBigInt = (bigInt *) malloc(sizeof(bigInt));
  int i;

  assert(aBigInt != NULL);
  aBigInt->value[0] = (uint_16) (initialValue & 0xFFFF);
  aBigInt->value[1] = (uint_16) (initialValue >> 16);
  aBigInt->length   = aBigInt->value[1] ? 2 : 1;

  for (i=2; i<MAX_WORDS; i++) {
    aBigInt->value[i] = 0x0000U; /* Zero extend */
  }
  return aBigInt;
```

```
}

void deleteBigInt(bigInt *aBigInt) {
  assert(aBigInt != NULL);
  free(aBigInt);
}

/*-------------------------------------------------------------------------*/
void bigIntSub(bigInt *A, bigInt *B) {
  int i,j;

  for (i = 0; i < LENGTH + 2; i++) {

    if (B->value[i] > A->value[i]) {
      j = i+1;
      A->value[j]--;
      while (A->value[j] == 0xFFFFFFFF)
        A->value[j++]--;
    }
    A->value[i] = A->value[i] - B->value[i];
  }
}

/*-------------------------------------------------------------------------*/
/* Inverse -modulus^(-1) mod B for odd modulus */
uint_16 bigIntInvMon(bigInt *modulus){
  unsigned int i;
  uint_32 x = 2, y = 1;
  uint_32 mZero = (uint_32)(modulus->value[0]);

  for (i = 2; i <= 16; i++, x <<= 1) {
    if (x < ((mZero * y) & ((x << 1) - 1))) {
      y += x;
    }
  }
  return (uint_16)(x - y);
}
/*-------------------------------------------------------------------------*/
```

```
char hexString[MAX_WORDS*4];
char hex[] = "0123456789ABCDEF";

char *bigInt2HexString(bigInt *aBigInt) {
  int i, j;
  uint_16 a;

  for (i=aBigInt->length-1, j=0; i>=0; i--,j+=4) {
    a = aBigInt->value[i];
    hexString[j]   = hex[(a >> 12) & 0X000F];
    hexString[j+1] = hex[(a >> 8 ) & 0X000F];
    hexString[j+2] = hex[(a >> 4 ) & 0X000F];
    hexString[j+3] = hex[(a      ) & 0X000F];
  }
  hexString[j] = 0x00;

  return hexString;
}

/*-------------------------------------------------------------------------*/
uint_16 char2int(char c) {
  switch(c) {
    case '0' : return 0; break;
    case '1' : return 1; break;
    case '2' : return 2; break;
    case '3' : return 3; break;
    case '4' : return 4; break;
    case '5' : return 5; break;
    case '6' : return 6; break;
    case '7' : return 7; break;
    case '8' : return 8; break;
    case '9' : return 9; break;
    case 'a' : return 10; break;
    case 'b' : return 11; break;
    case 'c' : return 12; break;
    case 'd' : return 13; break;
    case 'e' : return 14; break;
    case 'f' : return 15; break;
```

```
    case 'A' : return 10; break;
    case 'B' : return 11; break;
    case 'C' : return 12; break;
    case 'D' : return 13; break;
    case 'E' : return 14; break;
    case 'F' : return 15; break;
    default:
      printf("Invalid hex character: %c\n", c);
      exit(0);
  }
}

/*---------------------------------------------------------------------------*/
uint_16 quad2int(char *aCP) {
  return((char2int(aCP[0]) << 12) | (char2int(aCP[1]) << 8) |
    (char2int(aCP[2]) << 4)  | (char2int(aCP[3])));
}

/*---------------------------------------------------------------------------*/
void hexString2bigInt(char *theString, bigInt *b) {
  int len, lenm4;
  int i;
  char *aString;

  len = strlen(theString);
  lenm4 = (len % 4);
  if (lenm4) {
    lenm4 = 4 - lenm4;
  }
    aString = theString-lenm4;
  for (i=0; i<lenm4; i++) {
    aString[i] = '0';
  }
  len = strlen(aString);

  for (i=0; 4*i < len; i++) {
    b->value[i] = quad2int(aString+len-(4*i)-4);
  }
```

```
  b->length = i;
  for(; i < MAX_WORDS-1; i++) {
    b->value[i] = 0x0000U;
  }
}

/*-------------------------------------------------------------------------*/
int bigIntGTE(bigInt *A, bigInt *B) {
  int i;

  for (i=MAX_WORDS-1; i >= 0; i--) {
    if (A->value[i] > B->value[i]) return TRUE;
    if (A->value[i] < B->value[i]) return FALSE;
  }
  return TRUE;
}

/*-------------------------------------------------------------------------*/
int bigIntEQ(bigInt *A, bigInt *B) {
  int i;

  for (i=0; i<MAX_WORDS; i++) {
    if (A->value[i] != B->value[i]) return FALSE;
  }
  return TRUE;
}

/*-------------------------------------------------------------------------*/
uint_16 mZeroPrime;
bigInt *one;
bigInt *rSquared;

void setupMontgomery(bigInt *modulus, bigInt *rm) {
  mZeroPrime  = bigIntInvMon(modulus);
  rSquared    = rm;
}

/*-------------------------------------------------------------------------*/
```

```
int fiosInnerLoop(uint_16 *a, uint_16 bi, uint_16 *modulus, uint_16 *t, int
len, int i) {

   uint_32 sum1, sum2, carry=0;
   uint_16 u;
   uint_64 sum3_64;
   int j, retVal = 0;

/* First iteration of loop is unrolled so we can calculate u */
   sum1    = (((uint_32) a[0]) * ((uint_32) bi)) + ((uint_32) t[0]);
   u       = (uint_16)((sum1 & 0x0000ffff) * ((uint_32) mZeroPrime));
   sum2    = ((uint_32) modulus[0]) * ((uint_32) u);
   sum3_64 = (uint_64) sum1 + (uint_64) sum2;
   carry = (uint_32)(sum3_64 >> 16);

/* The rest of the loop iterations */
   for (j=1; j < len; j++) {
      sum1    = (((uint_32) a[j]) * ((uint_32) bi)) + ((uint_32) t[j]);
      sum2    =  (uint_32) modulus[j] * (uint_32) u;
      sum3_64 = ((uint_64) sum1) + ((uint_64) sum2) + ((uint_64) carry);
      t[j-1]  = (uint_16)(sum3_64 & 0x0000ffff);
      carry = (uint_32)(sum3_64 >> 16);
   }
   t[len-1]  = (uint_16)(carry & 0xFFFF);
   t[len]    = (uint_16)(carry >> 16);

   return(t[len]);
}

/*-------------------------------------------------------------------------*/
void hardware_mulmon(bigInt *a_l, bigInt *b_l, bigInt *modulus_l, bigInt *t_l)
{
   uint_16 *a       = a_l->value;
   uint_16 *b       = b_l->value;
   uint_16 *modulus = modulus_l->value;
   uint_16 *t       = t_l->value;
   uint_16 bj;
   int j;
   int my_length;
```

```
/* clear partial product accumulator */
  for (j=0; j < MAX_WORDS; j++) {
    t[j] = 0x0000U;
  }

  t_l->length = LENGTH;
  my_length = LENGTH;

  for (j=0; j<LENGTH; j++) {
    if (fiosInnerLoop(a, b[j], modulus, t, my_length, j))
      my_length++;
  }
}

/*-------------------------------------------------------------------------*/
void modularMultiply(bigInt *A, bigInt *B, bigInt *modulus, bigInt *product) {
  bigInt *aPrime = newBigInt(0);

  hardware_mulmon(A, rSquared, modulus, aPrime);

/* Sometimes the reduction is not complete */
  if (bigIntGTE(aPrime, modulus))
    bigIntSub(aPrime, modulus);

  hardware_mulmon(aPrime,B,modulus,product);

/* Sometimes the reduction is not complete */
  if (bigIntGTE(product, modulus))
    bigIntSub(product, modulus);

  deleteBigInt(aPrime);
}

/*-------------------------------------------------------------------------*/
int main(int argc, char **argv) {

  bigInt *a, *b, *c, *m, *rm, *e;
```

```
  FILE *aFile;
  int lengthBits;
  int test_number = 1;

  char theString[5001], tempChar;
  char *aString = theString+3;

  aFile = fopen(argv[1], "r");
  assert(aFile != NULL);

/* Note that the first line of the file is a seperator */
  while (fscanf(aFile, "%s\n", aString) != EOF) {

/* Size of the numbers in WORDS (16 bit quantities) */
     fscanf(aFile, "n=%s\n", aString);
     sscanf(aString, "%x", &LENGTH);

/* Modulus "m" */
     m = newBigInt(0);
     fscanf(aFile, "m=%s\n", aString);
     hexString2bigInt(aString, m);

/* Trap for the special cases where we miscalculate m->length above */
     if (m->length < LENGTH)
        m->length = LENGTH;

/* First mulitplicand "a" */
     a = newBigInt(0);
     fscanf(aFile, "a=%s\n", aString);
     hexString2bigInt(aString, a);

/* Second multiplicand "b" */
     b = newBigInt(0);
     fscanf(aFile, "b=%s\n", aString);
     hexString2bigInt(aString, b);
     fscanf(aFile, "k=%s\n", aString);

/* Length in bits - a sanity check from early testing */
```

```
     sscanf(aString, "%x", &lengthBits);

/* RM as calculated by Mathematica for this modulus */
     rm = newBigInt(0);
     fscanf(aFile, "rm=%s\n", aString);
     hexString2bigInt(aString, rm);

/* Result "c" */
     c = newBigInt(0);
     setupMontgomery(m, rm);

     modularMultiply(a, b, m, c);
     e = newBigInt(0);

/* Mathematica's answer: validate our solution */
     fscanf(aFile, "c=%s\n", aString);

     hexString2bigInt(aString, e);
     printf("Test %d ", test_number++);
     if(!bigIntEQ(c, e)) {
       printf(" failed\n");
       printf("Calculated: %s\n",bigInt2HexString(c));
       printf("  Expected: %s\n",bigInt2HexString(e));
     } else {
       printf(" passed\n");
     }

     deleteBigInt(a);
     deleteBigInt(b);
     deleteBigInt(m);
     deleteBigInt(c);
     deleteBigInt(e);
     deleteBigInt(rm);
   }
   fclose(aFile);
}
```